Testing without requirements?

REFSQ 2018, Utrecht, The Netherlands

Tanja E.J. Vos
Testing at the GUI Level

- GUI is where all functionality comes together
  - Interacts with the underlying code
  - The whole system can be executed by means of the GUI
  - Integration / System Testing
Testing at the GUI Level

- Most applications have GUIs
  - Computers, tablets, smartphones, ...
  - Even safety critical applications
Testing at the GUI Level

- Faults that arise at UI level are important.
- These are what your client finds.
- GUI tests from their perspective!
What is a GUI?

Contains graphical objects w, called **widgets**

- Menus, textboxes, buttons, scrollbars
Widgets form a hierarchy the *widget-tree*
Widgets have properties $p$ which have values $v$ at run-time.
GUI state

- The widget tree
- + the values of the properties of each widget

GUI action

- Users can exercise actions (click, type, drag, drop, ...)
- These cause a state change
What is GUI testing

- Sequences of GUI actions
  - Click, drag, drop, type
  - Provide inputs where needed (e.g., filling text fields)

- The test oracle
  - The correct states after execution of each action

Specify test sequences
Specify oracle

Together they test a requirement
What is GUI testing

Specify test sequences

Specify oracle

**Step 1**
Open MS Word

**Step 2**
Click on menu View

**Step 3**
Click on Media Browser

**Step 4**
Select a picture and drag into the document

After each step:
- No failure has occurred
- No error message has popped-up

After last step:
- The picture is in the doc
Step 1
Open MS Word

Step 2
Click on menu View

Step 3
Click on Media Browser

Step 4
Select a picture and drag into the document
Step 1
Open MS Word

Step 2
Click on menu View

Step 3
Click on Media Browser

Step 4
Select a picture and drag into the document
Step 1
Open MS Word

Step 2
Click on menu View

Step 3
Click on Media Browser

Step 4
Select a picture and drag into the document
Step 1
Open MS Word

Step 2
Click on menu View

Step 3
Click on Media Browser

Step 4
Select a picture and drag into the document
Manual testing

- Tedious
  - Executing the same clicks over and over again
- Tiresome and boring
  - Rerunning the same tests after changes to the SUT
  - Filling the same forms over and over again
  - Regression testing
- Error prone
- Costly
Need to automate
State of practice: make scripts

- Test sequences
- Oracles
- Develop Scripts
- Execute
- Maintenance

Capture & Replay
Visual testing
Capture & Replay

- Tools Captures user interaction with the UI
- Records a script
- That can be automatically Replayed

Examples
- Open source
  - Selenium
  - Abbot
  - ....
- Commercial
  - QF-Test
  - Rational Functional /Robot Tester (IBM)
  - ....
Capture & Replay
Capture & Replay

Advantages
- Simple and easy

Disadvantages
- Scripts break as GUI changes
- Maintenance problem

These are huge problems
- GUls change all the time
- Requirements too!
Visual testing (VGT)

- Based on image recognition
Visual testing

- Easy to understand
- Hardly no programming skills needed
- Solves part of maintenance problem
  - Robust against some changes
  - But not all
    - Move Media Browser within same menu: YES
    - Move Media Browser to another menu: NO
    - Change the icon: NO
- Studies show maintenance still an issue
Our contribution: Test*

- **Scriptless**
  - What is not there does not need to be maintained

- Departs from random testing
  - Immediately start testing without requirements

*OH YEAH!
How?

Start SUT

Start TESTAR

Scan SUT for current GUI state

Update

Check online state oracles

Failure detected

Executable test sequence leading to a failure

Check stop criteria

yes

Stop SUT

Execute action

Select action

Derive (+filter) actions

Query for offline oracles

GUI state graph database

Test*
Current state and actions
Select action

VLC media player menu:
- File...
- Multiple Files...
- Folder...
- Disc...
- Network Stream...
- Capture Device...
- Location from clipboard
- Recent Media
- Playlist to File...
- Convert / Save...
- Stream...
- Quit at the end of playlist
- Quit
Execute to go to new state
How?

1. Start SUT
2. Execute action
3. Select action
4. Derive (+filter) actions
5. Check stop criteria
6. Check online state oracles
7. Scan SUT for current GUI state
8. Query for offline oracles
9. Executable test sequence leading to a failure
10. Failure detected
11. Stop SUT

Start TESTAR

Test*
READY

PATH to SUT
POWERPNT.exe
SET

undesired processes

undesired actions
We can start automated testing

- Immediately (minimal set-up)
- No scripts
- No maintenance here
  - The widget tree is extracted in each new state
  - If the state is different, so is the widget tree
100% Automated online oracles

Verdict oracle_Crash (State state){
    if(!state.get(IsRunning,false))
        return new Verdict("System crashed!");
}

Verdict oracle_Responsiveness (State state){
    if(state.get(NotResponding, true))
        return new Verdict("System not responding!");
}

- Crashes
- Hangs
• Online oracles for suspicious titles and outputs
• Specify them with a regular expression
Regular expressions
Verdicts oracle_SuspiciousTitles(State state) {
    verdicts = new Verdicts();
    String regEx = settings().get(SuspiciousTitles);

    // search all widgets for suspicious titles
    for (Widget w : state) {
        String title = w.get(Title, "");
        if (title.matches(regEx)) {
            verdicts.add(new Verdict("suspicious title.."));
        }
    }
    return verdicts;
}
ClaveiCon

- Spanish SME
- ERP system
- Written in Visual Basic
- Microsoft SQL Server 2008 database
- Targets the Windows operating systems.

<table>
<thead>
<tr>
<th></th>
<th>TESTAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preparation</td>
<td>26 hour</td>
</tr>
<tr>
<td>Testing</td>
<td>91 hour</td>
</tr>
<tr>
<td>Post testing</td>
<td>1,5 hour</td>
</tr>
<tr>
<td>Critical faults</td>
<td>10</td>
</tr>
</tbody>
</table>
SOFTEAM

- French and large company
- Backend system for virtualization
- Web GUI
- We could re-inject existing faults

\[ \text{FDR} = \frac{\text{num of Faults found}}{\text{num of injected Faults}} \times 100\% \]

<table>
<thead>
<tr>
<th></th>
<th>TESTAR</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preparation</td>
<td>40 hour</td>
<td>36 hour</td>
</tr>
<tr>
<td>Testing</td>
<td>77 hour</td>
<td>1 hour</td>
</tr>
<tr>
<td>Post testing</td>
<td>3,5 hour</td>
<td>2 hour</td>
</tr>
<tr>
<td>FDR</td>
<td>61%</td>
<td>83%</td>
</tr>
<tr>
<td>Code coverage</td>
<td>70%</td>
<td>86%</td>
</tr>
</tbody>
</table>
Cap Gemini/ProRail

- Dutch cooperation
- Web GUI
- System for managing the assignment of train platforms

<table>
<thead>
<tr>
<th></th>
<th>TESTAR</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preparation</td>
<td>44 hour</td>
<td>43 hour</td>
</tr>
<tr>
<td>Testing</td>
<td>51 hour</td>
<td>6 hour</td>
</tr>
<tr>
<td>Post testing</td>
<td>5 hour</td>
<td>2 hour</td>
</tr>
<tr>
<td>Critical faults</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Functional coverage</td>
<td>80%</td>
<td>73%</td>
</tr>
</tbody>
</table>
Beside these Test

- Microsoft office suite
- Bitrix 24
- Test the test tool TESTONA (eclipse based)
- Over 10 web applications of Spanish companies
- 12 students currently working on it
- Several companies doing proof of concepts
How does it change testing?
How?

1. **Start SUT**
2. **Scan SUT for current GUI state**
3. **Check online state oracles**
4. **Query for offline oracles**
5. **Update**
6. **Executable test sequence leading to a failure**
7. **Check stop criteria**
8. **Derive (+filter) actions**
9. **Select action**
10. **Execute action**
11. **Stop SUT**
12. **Start TESTAR**
Verdicts oracle_ImagesWAI(State state) {
    verdicts = new Verdicts();
    for(Widget w : state){
        Role role = w.get(Tags.Role);
        if (role.equals("UIAImage") && title.isEmpty())
            verdicts.add(new Verdict("Not all images have an alternate textual description");
    }
    return verdicts;
}
Offline oracles: Query the graph database
Application/Domain specific oracles

Need to be programmed/specified

We cannot avoid making oracles manually

TESTAR shares this problem with ALL automated approaches
How does it change testing?
How is the test effort distributed

- Specify test case
- Specify oracle
- Capture or develop test scripts
- Maintainance
- Automated test execution
How can Test* change testing?

- Specify test case
- Specify oracle
- Capture or develop test scripts
- Maintainance
- Automated test execution
Random testing

“Valuable test case generation scheme”

“In the 70s

“Necessary final step in the testing activities”

“Probably the poorest testing method”
Use partition testing

Use domain knowledge of the SUT to partition
Group together similar test cases
Choose one
Random testing

Duran and Ntafos (1984): simulation and experiments showing random better than systematic partition testing.

Hamlet and Taylor (1988): more experiments showing the same
Random testing

Why do random testing and systematic testing seem to be almost on par?

What are the properties of random testing?

When is random testing more effective than partitioning and the other way around?
A Probabilistic Analysis of the Efficiency of Automated Software Testing

Marcel Böhme and Souryna Paul

Abstract—We study the relative efficiency of the random and systematic approaches to automated software testing. Using a simple but realistic software model, we propose a general model for software testing and derive exact sampling examples for random (u) and systematic (s) testing, where such sampling is associated with a sampling cost. 1 and v are values of long-term observability. The most important goal of software testing is (u) discovering a maximal number of errors at once (a green light). (s) improves the probability of a program's correctness and (r) discovers a maximal number of errors in a green light. For both (u) and (s), we show that there is a bound on whether the probability of a program's correctness is increased beyond which it performs better than (s), on this average. Moreover, (u), (s), and (r) depend on the random element in the test suite. Depending on this element, it is shown that the efficiency of (u) can be fixed to an upper limit. Using these results we design a hybrid strategy that starts with (u) and switched to (s), when it is exposed to discover more defects per time in our experiments we find that (u) performs similarly in better than the most efficient in both and that (s), may need to be significantly larger than our bounds to keep its effectiveness over (r).

Index Terms—Random testing, systematic testing, error-based partitioning, efficient testing, testing theory.

1 INTRODUCTION

Efficiency is an important property of software testing. Potentially even more important than effectiveness because complete software testing is not even feasible, widely distributed programs for many years [2], [3], developers are looking for automated techniques to gain confidence in their program's correctness. The most effective way to improve confidence in the program's correctness is for all inputs in called program verification. However, due to static analysis and other problems, the applicability of verification remains limited to a few lines of code. Now, software testing trades this effectiveness for efficiency. It allows one to gain confidence in the program's correctness with a small test input that is computationally expensive. automated testing is an efficient way to improve confidence in the program's correctness for an increasing set of inputs. Yet, most research of software testing has mainly focused on effectiveness:

The most effective testing technique reveals a maximal number of defects and reaches a maximum degree of confidence in the correctness of programs.

Only very recently we started to investigate its efficiency.

The most efficient testing technique is the one that is successful in the shortest time or the one that generates the most effective test suite with the given time budget.

Using a simple set of assumptions, we construct a general model of software testing, define testing strategies, where each generated test input is subject to a cost, and cost our efficiency analysis as a problem in probability theory.

We model the testing problem as an exploration of error-based error-partitions, supported for a program there exists a partitioning of its input space into homogeneous subsets [1], [2]. For each such partition, either all inputs reveal an error or none of the inputs reveal an error. The number and "size" of such error-based partitions can be arbitrary and need not be bounded. Assuming that it is unknown with certainty whether or not a partition reveals an error, the problem of software testing is to sample each partition in a systematic fashion to gain confidence in the correctness of the program.

A testing technique samples the program's input space.

We say that a partition D is discovered when D is sampled for the first time. The sample of a partition D reveals whether or not partition D reveals an error. Efficiently, the testing input becomes a witness for the error-revealing property of D. A testing technique achieves the degree of confidence 

\[ P \] when at least \( \alpha \) percent of the program inputs result in discovered partitions. Hence, if none of the discovered partitions reveals an error, we can be certain that the program works correctly at least for a percent of its inputs.

For our efficiency analysis, we consider two strategies: random testing that is oblivious of error-based partitions and systematic testing that samples each partition exactly once. Random testing is not uniform at random and might sample some partitions several times and some not at all. Specifically, we show that for \( R \) the number and size of partitions discovered decays exponentially over time. Systematic testing samples each error-based partition exactly once and thus strictly increases the established degree of confidence.

We model a systematic testing technique \( D \), that chooses the order in which partitions are discovered uniformly at random and shows that number and size of partitions discovered grows linearly over time. Note that our theoretical results are relevant to computer science.

"Even the most effective testing technique is inefficient compared with random testing if generating a test case takes relatively too long!"
For automated GUI testing.....

- Generating test case is:
  - Specification
  - Capture (or automate with script)
  - Maintenance!!

- And random selection gave us quite good results on the software we tested .......

- Can we do better?
How can we find more faults?

- Some test cases might be more likely to reveal faults
- Don´t pick at random, but try to optimize criteria!
- What criteria?
Where can we find faults?

- Surrogate measures
- We cannot measure % of faults found
- We measure something we believe, hope or have shown to be correlated to that attribute.
- Coverage
- Diversity
- Novelty

Let the testing tool learn by itself how to test better!!
Surrogate measures

- as many **different actions** as possible? **Q-learning**
- make **large call trees**? **Ant colonies**
- visit as **many different states** as possible? **Evolutionary algorithms**
- make **long sequences**? **Evolutionary algorithms**
- find **novel states**?
- We need to investigate many more
Machine Learning (Q-learning)

- sets $S$ of possible states
- sets $A$ of possible actions
- description $T$ of the effect of action in a state

$$T: S \times A \rightarrow S$$

- state $s$ then select an action from $a \in A$ that causes a transition to a next state $s'$
- reward function $R: S \times A \rightarrow \mathbb{R}$

find a policy $\pi$ which maximizes the reward by selecting an appropriate action in each state
Rewards

- Set $S$ of possible states the SUT can be in
- For all $s \in S$, we have sets $A_s \subseteq A$ of actions
- We focus is on exploration of the GUI
- We reward actions $a$ with low execution count $ec$

$$\forall s \in S, a \in A_s: R(s, a) = \begin{cases} R_{\text{max}}, & ec(a) = 0 \\ \frac{1}{ec(a)}, & \text{otherwise} \end{cases}$$
\textbf{Q-learning algorithm}

\textbf{Require:} \( R_{\text{max}} > 0 \) \hspace{1em} /* reward for unexecuted actions */

\textbf{Require:} \( 0 < \gamma < 1 \) \hspace{1em} /* discount factor */

1: \textbf{begin}
2: \quad \text{start SUT}
3: \quad \forall (s, a) \in \mathcal{S} \times \mathcal{A} : \quad Q(s, a) \leftarrow R_{\text{max}}
4: \quad \text{initialize } s \text{ and available action } A_s
5: \quad \textbf{repeat}
6: \quad \hspace{1em} a^* \leftarrow \max_a \{ Q(s, a) | a \in A_s \}
7: \quad \hspace{1em} \text{execute } a^*
8: \quad \hspace{1em} \text{obtain state } s' \text{ and available actions } A_{s'}
9: \quad \hspace{1em} Q(s, a^*) \leftarrow R(s, a^*) + \gamma \cdot \max_{a \in A_s} Q(s', a)
10: \quad \hspace{1em} ec(a^*) ++
11: \quad \hspace{1em} s \leftarrow s'
12: \quad \textbf{until} \text{ stopping criteria met}
13: \quad \text{stop SUT}
14: \textbf{end}
Ant Colony Optimization

- Collectively ants can solve complex tasks
- Ants communicate using pheromones
  - They lay this on their path
  - Pheromone trail strength accumulates when multiple ants use a path
  - Other ants go where there is good pheromone strength
Ant Colony Optimization

- We have a population of ants
- Set of choices $C$ (= actions)
- The ants generate trails (= test sequences)
- By choosing $c_i$ according to pheromone values $p_i$ (= selection criteria)
- Choices (= actions) that appear in “good” trails (= max call tree) accumulate pheromones
Evolve action selection rules

Crossover → Selection → Evaluation → Terminate? → Mutation

Test Outputs
- Coverage
- Failures
- States
- Etc.

Calculate Fitness

Test* → Individuals
Action selection rules

IF-THEN

true

pick

pick

any

num_buttons

IF-THEN-ELSE

>

pick

num_textfields

any

button

pic

textfield

1st
Crossover
Mutation
Evolve action selection rules

- Crossover
- Selection
- Evaluation
- Terminate?

Test Outputs
- Coverage
- Failures
- States
- Etc.

Calculate

Individuals

Test*
Let the testing tool learn itself how to test!
  - Use different machine learning algorithms (action selection/oracles)
  - Define more surrogate measures

Learn from what the tool tests
  - Show that surrogate measures work
  - Relate them to (type of) failures
  - Extract models to aid exploratory testing
  - Improve visualisation

More formal testing theory
  - Know better whether we have done well

Reduce the human oracle cost:
  - Automate as much as possible all other test tasks
  - Make it as easy as possible for the tester
TESTAR Training @ TNO

- 15 and 16\textsuperscript{th} of May 2018
- TNO in Groningen
- Training, hands-on and helpdesk!
- Interested?
- Send me an email.
• **email:** info@testar.org
• **web:** http://www.testar.org/
• LIKE IT on: http://facebook.com/tool.testar
• **telephone/whatsapp:** +34 690 917 971